# *Out of the Sandbox: Third Party Validation for Java Applications*

I. Jermyn, F. Monrose, P. Wyckoff
Department of Computer Science
New York University
251 Mercer Street, NYC, NY, 10012

## Abstract

Java's security allows a user to import and run applets from the Web without undue risk to the user's machine by restricting the applet's actions to a "sandbox", an area of the web browser dedicated to that applet. The sandbox model is critical to Java's success and its promise of truly network-oriented computing. Applications running in the sandbox can only access local and network resources through a limited number of trusted mechanisms. This model gives users the advantage of easy distribution of software while protecting them from potentially malicious applications, but can be too restrictive at times. To address the need to extend the flexibility of the "all-or-nothing" approach, the concept of *object signing* was introduced to the Java model. Access to resources outside the sandbox is granted based on a user-defined policy consisting of a list of code signers and the type of access each signer is allowed. While this policy works well for restricting access to "trusted" code from a well-known signer, non-malicious, entertaining or educational code from individual programmers or small businesses is cast out, unless privileges for *each* signed authority are incorporated into the user's access matrix. In this paper we propose that the privileges required by code from unknown sources be verified and signed by a single trusted third party and present an infrastructure to facilitate this proposal. We then describe a parallel application built on top of the Charlotte [2] parallel processing system and an order inventory database application as exemplars of this approach.

## 1 Introduction

Consider the following scenario. Alice wants to run her parallel processing code on Bob's machine. She has written her code to take advantage of a trusted code base from BigDeveloper Inc. Her code is non-malicious, and requires the full privileges of classes provided within BigDeveloper's software development kit. Unfortunately Bob has no reason to trust Alice, and refuses to grant her code the necessary privileges. Due to the nature of Java's extended stack introspection, in which the security model assigns runtime privileges based on the consensus voting rule [6], Alice's parallel processing code is never correctly executed. Alice cannot obtain higher privileges for her code since these privileges are assigned by Bob on the basis of the principal responsible for the code source, and Alice runs a small operation that is virtually unknown.

This scenario is not unusual. Users are continually encountering code on the Web that cannot be trusted, and in the absence of other information are forced for their own safety to deny it the privileges necessary for its execution. In essence, this means that full use of the Web is denied to both users and the developers providing them with a trusted computing base. Even if Alice and her many counterparts obtained digital IDs and had their code signed by certifying authorities, Bob would still face the tedious administrative task of assigning each principal to a particular entry in his access matrix.

We are proposing a certification procedure based on a single trusted third party who is given a set of privileges on the users' machine, and who can guarantee varied levels of safety of (in this case) Alice's code if it conforms to certain restrictions. The benefits are two-fold. Firstly, the user is no longer responsible for the administration of the security privileges assigned to each code source. In addition, smaller code sources can utilize the full power of trusted libraries without danger of harm to the user.

In the next few sections we discuss some issues pertaining to the implementation of code signing in the two most prominent browsers. We then discuss our proposal for a third party verifier, outline a method for the third party to verify the code, and discuss two applications that benefit from our proposals.

## 2 Object Signing

Object signing is a mechanism which allows users to obtain reliable information about downloaded classes by using standard cryptographic techniques such as one-way hash functions and message authentication checks. Reliable software distribution over the Internet poses many serious security problems, most notably ensuring safety, integrity and accountability. Object signing has been proposed as a solution to the latter two problems. Integrity means that the byte stream has not been interfered with and altered during transit, and accountability means that the code should be associated to a particular identifiable principal.

The use of object signing in Java facilitates operations beyond the defined limits of the sandbox environment. Such extended operations include file access and establishing arbitrary network connections. The burden of determining to what granularity these normally restricted accesses are granted, and which signers are permitted access, is left entirely up to network administrators and users. While this approach permits a fine-grained continuum of access privileges from relatively innocuous operations [4], creating and managing a user's access matrix can be somewhat burdensome.

The Java object signing model is based on the notion of capabilities [3], which has existed in the security component of many novel systems such as Taos [7] and Amoeba [5] for quite some time. A capability is a pointer to a controlled system resource that cannot be duplicated, thus protecting the resource from misuse. A program that wishes to use a controlled resource must do so through a capability, but the ability to use a capability needs to be explicitly assigned to the requesting program, either during its initialization or by a call to another capability.

In Java, object signing reduces to a capability based system where digital signatures which accompany an applet are represented as *principals*, resources are represented by *targets*, and the *privileges* associated with a principal represent the authorization for a principal to access a specific target [4]. These signatures represent endorsements of the code by the signer, asserting that the code is not malicious and behaves as advertised [6].

The extended stack introspection mechanism implemented by both Netscape and Microsoft uses digital signatures to match pieces of incoming byte code to principals, and consults a policy engine to determine which system targets should be enabled for a particular principal. Microsoft's approach to handling digital signatures is to allow only one signature on each piece of code, and each signature contains a list of targets to which the signer thinks the code should be given access. By contrast, Netscape's approach allows for multiple signers, with no a priori mention of targets.

In Netscape, because of the possibility of multiple signers for a particular class, the policy engine uses a consensus voting rule to determine which privileges are granted on behalf of the signers. Intuitively, consensus voting means that one negative vote can force access to be forbidden, while at least one positive vote (and no negative votes) is required in order to allow access to a target [6]. When a privilege is enabled, an annotation is recorded on the call stack of the requesting thread, so that by querying the call stack for these annotations, future calls to the policy engine are avoided. However, when a method call crosses classes with differently signed principals, the enabled privileges are hidden. Therefore, for Alice's code to move beyond the sandbox and execute correctly with, for example, "networking" privileges, the signature of a more trusted principal is needed.

## 3 Verifier

In this section, we describe the third party verification method and present one possible framework that supports its implementation for a restricted but large class of applications.

Third party verification is a method by which Java's security model is extended (without changing the JVM implementation) to allow trusted third parties to grant well-behaved applications the privileges necessary to access restricted system targets. In this way, well-behaved applications from small companies can be granted access to targets outside the Java sandbox without having to explicitly trust the primary principal associated with the code source. Trust is placed in the third party who is delegated the responsibility for verifying that the code is well-behaved.

The proposed verification process proceeds as follows. Alice sends her code and a request to be granted privileges for target(s) $\{p\}$ to the third party verifier [1] The verifier checks whether Alice's code attempts to perform any malicious operations with the $\{p\}$ targets, and if not, signs her code with the signature which grants privileges for $\{p\}$. Alice's code may be executed by any user that grants the third party the

[1] Our implementation requires that the list of requested targets precedes the code, as is the case with Microsoft's implementation of object signing.

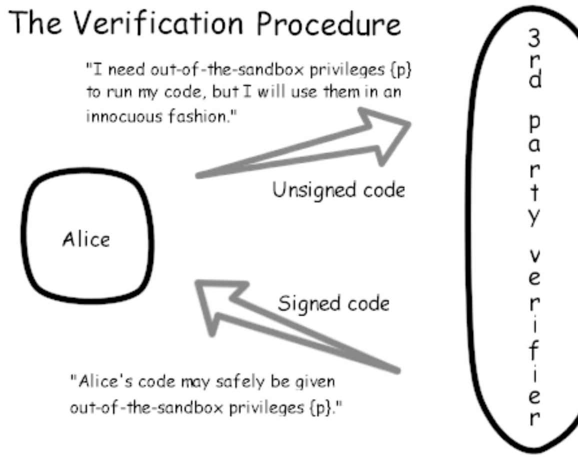
The Verification Procedure
"I need out-of-the-sandbox privileges {p} to run my code, but I will use them in an innocuous fashion."
Unsigned code
Alice
Signed code
"Alice's code may safely be given out-of-the-sandbox privileges {p}."
3rd party verifier

Figure 1: Alice sends her code to the third party verifier.

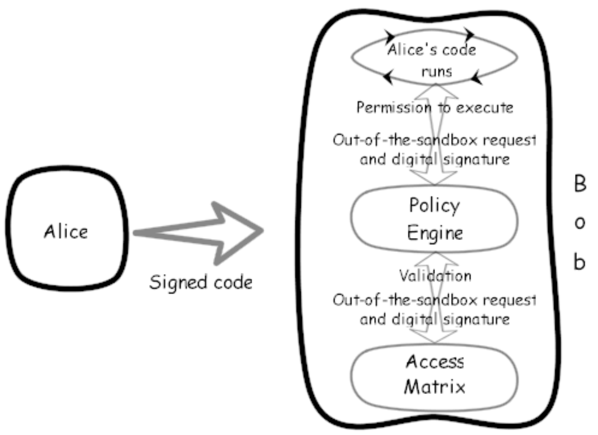
Alice's code runs
Permission to execute
Out-of-the-sandbox request and digital signature
Policy Engine
Validation
Out-of-the-sandbox request and digital signature
Access Matrix
Alice
Signed code
Bob

Figure 2: Bob downloads Alice's code from the Web and runs it.

proper privileges. This is depicted schematically in Figure 1. It is important to note that the third party verifier does not need to verify the overall safety of Alice's code: the use of restricted targets not in {p} is controlled by the standard Java security manager. Figure 2 shows Alice's code being executed on Bob's machine.

The third party verification method is only useful when there is a method of verifying that an application will use the targets that it has requested in an innocuous fashion. In the rest of this section, we propose a framework that facilitates this verification and show how a simple order inventory application and a parallel application are supported within this framework.

The framework we propose is simple: a trusted source, BigDeveloper Inc, creates a high-level library that accesses restricted targets in such a way that if the library is used in a prescribed manner, the use will be innocuous. The application developer, Alice, then writes her code using the library in the prescribed manner and requests that the third party verifier attest to this so that the end-user Bob will grant her code the privileges necessary to use the library. The third party verifier checks that Alice's code does in fact use the library in the prescribed manner and in addition that the code does not abuse any privileges it may acquire by deliberately or accidentally accessing restricted targets without using the library. In this way, restricted targets can be made available to application builders while still protecting the end-user's execution environment.

The library must be high-level enough that it cannot be used for purposes other than it is intended. For example, a socket library that allows Alice to open a socket, and read and write byte streams from the socket is too low-level because she could use it to perform third party attacks. On the other hand, if the library provides high-level shared memory operations and is such that it can only communicate with other instances of the same library (as is the case, for example, with Charlotte as discussed below), then it is safe to allow Alice to use it.

We denote the targets that the library uses as $L_t$ and the manner in which the library must be used as $L_m$. We now describe a parallel processing application built using a parallel library, and a simple inventory and order database application built using a database library.

We use Charlotte [2] as our model of a parallel processing library. Charlotte [2] leverages Java and the Calypso [1] programming model to provide a powerful metacomputing substrate for the WWW. The programming model is simple, consisting of parallel loops and CREW distributed shared memory. Load balancing and fault tolerance are provided transparently by the runtime system, and the Charlotte library is written entirely in Java. The Charlotte system we describe is slightly modified so that it meets the requirements of a library in our framework.

The Charlotte library communicates with a Charlotte Manager through a socket. The Java socket library is the only restricted parameterized target that the Charlotte library uses (i.e., $L_t = \{NetworkConnect\}$). In Charlotte, all application communication is done through shared memory "read"s and "write"s. Although the application can cause data to be read and written to and from the Charlotte socket connection, it is greatly restricted because at a low-level, that data is packaged and sent in a Charlotte specific way. For example, when the connection is first established the String " am Alice's Charlotte Worker"is the first data sent and the library will throw an exception if the String " am Alice's Charlotte Manager" is not received.

The manner in which the use of the library is restricted, expressed by $L_m$, states that the ratio of communication to computation must be low in order to prevent denial of service attacks.

The third party verification for an application that uses Charlotte simply consists of checking that the application does not use sockets except via the Charlotte library, and that the amount of communication is much less than the amount of computation [2].

Our second example application is a simple ordering and inventory control database for small companies who want to distribute applets to current and prospective customers. In this type of application, only simple user input forms and database transactions are necessary. A simple database interface library is used to allow these applications to access their databases.

The library uses a parameterized system target and $L_t = \{UniversalNetworkConnect\}$. We restrict its use so that it may only connect to machines in the application developer's domain. In this way, we ensure that Alice does not try to attack her competitor's databases, but she is free to access any of her databases around the world. Because Alice is only allowed to connect to databases within her company's domain, we can allow her to perform any operation as many times as she likes with the library. We could also allow Alice to read and write to a single file in a specified directory using this method.

## 4 Conclusion

In this paper, we proposed third party verification as a method to permit applets to move beyond the confined limits of the Java security sandbox. We proposed an implementation framework of third party verification based on having applications use well-defined, high-level libraries to access restricted targets. In the future, we plan on performing a more rigorous examination of the proposed model, and to explore alternative implementation frameworks.

[2]In our prototype implementation, we restrict loops to depend on constants and do not allow recursion; thus we can check this ratio

### Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Material Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grant number CCR-94-11590; and by the Intel Corporation.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

## References

[1] Arash Baratloo, Partha Dasgupta, and Zvi Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. *4th IEEE International Symposium on High Performance Distributed Computing*, 1995.

[2] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. *9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[3] H. Levy. Capability-Based Computer Systems. *Digital Press*, 1984.

[4] McGraw and Felten. Understanding the keys to Java security:the Sandbox and authentication. *JavaWorld*, 2(5), May 1997.

[5] A.S Tanenbaum, S.J Mullender, and R. Van Renesse. Using sparse capabilities in a distributed operating system. *6th International Conference on Distributed Computing Systems*, pages 558–563, 1986.

[6] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward Felton. Extensible Security Architectures for Java. *16th Symposium on Operating System Principles*, 1997.

[7] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos Operating System. *TOCS*, 12(1):3–32, February 1994.